# En Scala Tutorial
## för Javaprogrammerare

Version 1.3
July 13, 2010

**Michel Schinz, Philipp Haller**
**Översättning: Nils Fredrik Karlsson**

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

# 1 Inledning

Detta dokument ger en snabb introduktion till språket Scala och dess kompilator. Den är till för personer som redan har erfarenhet av programmering sedan tidigare och vill ha en överblick av vad de kan göra i Scala. Det antas att läsaren har en grundläggande kunskap av objektorienterad programmering, speciellt i Java.

# 2 Ett första exempel

Som ett första exempel skall vi skriva det klassiska programmet *Hello World*. Det är inte särskilt häpnande, men demonstrerar användningen av de verktyg som finns i Scala utan att man behöver veta något om språket i förväg. Så här ser koden ut:

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

Strukturen av programmet bör vara bekant för Javaprogrammerare: det består av en metod main som tar emot ett fält av strängar som kommandoradsargument. Metodens innehåll består av ett enda anrop till den fördefinierade metoden println med den vänliga hälsningen som argument. Metoden MAIN returnerar inget värde (det är en procedurmetod). Därmed är det inte nödvändigt att deklarera en returtyp.

Vad som är mindre bekant för Javaprogrammerare är att deklarationen av **object** innehåller metoden main. En sådan deklaration introducerar vad som brukar kallas ett singleton objekt, d.v.s. en klass med en enda instans. Deklarationen ovanför deklarerar därmed både en klass, kallad HelloWorld, och en instans av klassen, även den kallad HelloWorld. Denna instans skapas vid behov, d.v.s. första gången den används.

Den skarpsinniga läsaren kanske har märkt att metoden main inte är deklarerad att vara static. Detta är för att statiska medlemmar (metoder eller fält) inte existerar i Scala. Istället för att definiera statiska medlemmar, så deklarerar Scalaprogrammeraren dessa medlemmar i singleton objekt.

## 2.1 Kompilera exemplet

För att kompilera exemplet använder vi scalac, Scalakompilatorn. scalac fungerar som de flesta kompilatorer: den tar en källfil som argument, tillsammans med eventuella argument, och producerar en eller flera objektfiler. Objektfilerna den producerar är vanliga Java klassfiler.

Om vi sparar programmet ovanför i en fil kallad HelloWorld.scala, så kan vi kompilera

den genom att skriva ut följande kommando (större-än symbolen ’>’ representerar kommandoprompten och skall alltså inte skrivas ut):

```
> scalac HelloWorld.scala
```

Detta kommer generera några klassfiler i den nuvarande katalogen. En av dessa kommer kallas HelloWorld.**class** och innehåller en klass som kan exekveras direkt m h a kommandot scala, som följande sektion kommer visa.

### 2.2 Köra exemplet

När programmet är kompilerat kan Scala programmet köras med kommandot scala. Dess användning är mycket lik den för kommandot java som används för att köra Javaprogram, och accepterar även samma argument. Ovanstående exempel kan exekveras m h a följande kommando, vilket producerar det förväntade resultatet:

```
> scala -classpath . HelloWorld

Hello, world!
```

## 3 Interaktion med Java

En av Scalas starka sidor är att det gör det väldigt enkelt att interagera med Javakod. Alla klasser från paketet java.lang importeras automatiskt, medan andra behöver importeras explicit.

Låt oss se på ett exempel som demonstrerar detta. Vi vill erhålla och formatera det nuvarande datumet enligt konventioner använda i ett specifikt land, t.ex. Frankrike [1].

Javas klassbibliotek definierar kraftfulla hjälpklasser, såsom Date och DateFormat. Då Scala interagerar sömlöst med Java, så behöver man inte implementera motsvarande klasser i Scalas klassbibliotek – vi behöver helt enkelt bara importera klasser av korresponderande Javapaket:

[1] Andra regioner såsom den fransktalande delen av Schweiz använder samma konventioner.

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

object FrenchDate {
  def main(args: Array[String]) {
    val now = new Date
    val df = getDateInstance(LONG, Locale.FRANCE)
    println(df format now)
  }
}
```

Scalas importfunktionalitet liknar den för Java, fast är betydligt mer kraftfull. Ett flertal klasser kan importeras från samma paket genom att innesluta dem med måsvingar, som vi gjorde på första raden. En annan skillnad är att när vi importerar alla namn i ett paket eller i en klass, så används ett understreck (_) istället för en asterisk (*). Detta för att asterisken är en giltig identifierare (t.ex. för metodnamn), som vi kommer se senare.

Importinstruktionen på tredje raden kommer således importera alla medlemmar av klassen DateFormat. Detta gör den statiska metoden getDateInstance och det statiska fältet LONG synliga.

Inuti metoden main skapar vi först en instans av Javaklassen Date som av standard innehåller det nuvarande datumet. Därefter definierar vi ett datumformat med den statiska metoden getDateInstance som vi importerat sedan tidigare. Till sist skriver vi ut det nuvarande datumet formaterat enligt den lokaliserade instansen av DateFormat. Denna sista rad visar på en intressant egenskap hos Scalas syntax. Metoder som tar ett argument kan användas med infix syntax. Alltså, uttrycket

```
df format now
```

är bara ett annat, mindre verbost sätt att skriva uttrycket

```
df.format(now)
```

Detta kan verka vara en mindre syntaktisk detalj, men det har viktiga konsekvenser, varav en av dem kommer undersökas i nästa sektion.

För att avsluta denna sektion om integration med Java bör noteras att det även är möjligt att ärva från Javaklasser och implementera Javagränssnitt direkt i Scala.

# 4 Allting är objekt

Scala är helt och hållet objektorienterat, på så sätt att *allting* är ett objekt, inklusive tal eller funktioner. Det skiljer sig från Java på så sätt, då Java skiljer på primitiva typer (såsom `boolean` och `int`) och referenstyper, och gör det inte möjligt att manipulera funktioner som värden.

## 4.1 Tal är objekt

Då tal är objekt innebär det även att de har metoder. Faktum är att ett aritmetiskt uttryck som följande:

```
1 + 2 * 3 / x
```

består uteslutande av metodanrop, för att det motsvarar följande uttryck, som vi såg i föregående sektion:

```
(1).+(((2).*(3))./(x))
```

Detta betyder även att `+`, `*`, etc. är giltiga identifierare i Scala.

Parenteserna kring talen i den andra versionen är nödvändiga då Scalas lexer använder den längsta matchningsregeln för tokens. Det skulle därmed bryta ut följande uttryck:

```
1.+(2)
```

i de mindre tokens `1.`, `+`, och `2`. Anledningen till att denna tokenisering används är för att `1.` är en längre giltig matchning än `1`. Således tolkas `1.` som literalen `1.0`, vilket gör det till en `Double` istället för en `Int`. Att skriva uttrycket som:

```
(1).+(2)
```

förhindrar `1` från att tolkas som en `Double`.

## 4.2 Funktioner är objekt

Det som kanske är mest överraskande för Javaprogrammeraren är att funktioner även är objekt i Scala. Det är således möjligt att skicka funktioner som argument, att lagra dem i variabler, och att returnera dem från andra funktioner. Denna förmåga att manipulera funktioner som värden är en av hörnstenarna av en mycket intressant programmeringsparadigm kallad *funktionell programmering*.

Som ett simpelt exempel på varför det kan vara användbart med funktioner som värden kan vi tänka oss en timer funktion vars syfte är att utföra någon händelse varje sekund. Hur skickar vi in händelsen som skall utföras? Det mesta logiska vore i form av en funktion. Detta väldigt enkla sätt att skicka funktioner bör vara bekant för de flesta programmerare: det används ofta för användargränssnitt, där man

registrerar s.k. callback-funktioner, vilka anropas när någon händelse uppstår.

I följande program anropas timer-funktionen oncePerSecond, och tar emot en callback-funktion som argument. Typen av denna funktion skrivs som () => och är typen av alla funktioner som inte tar emot några argument och inte heller returnerar något (typen Unit liknar void i C/C++). Funktionen main i programmet anropar helt enkelt timer funktionen med en callback som skriver ut sekvensen på terminalen. M a o skriver programmet ut meningen ”time flies like an arrow” varje sekund oavbrutet.

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies() {
    println("time flies like an arrow...")
  }

  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
  }
}
```

Notera att för att kunna skriva ut strängen använde vi den fördefinierade metoden println istället för den från System.out.

### 4.2.1 Anonyma funktioner

Medan detta program är enkelt att förstå, så kan det förfinas. Notera för det första att funktionen timeFlies endast definieras med syfte att senare skickas till funktionen oncePerSecond. Att namnge denna funktion, som endast används en gång, kan verka onödigt, och det vore trevligt att kunna konstruera denna funktion när vi skickar den till oncePerSecond. Detta är möjligt i Scala m h a *anonyma funktioner*, vilka är just detta: funktioner utan ett namn. Den ändrade versionen av vårt timerprogram använder en anonym funktion istället för timeFlies och ser ut så här:

```
object TimerAnonymous {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("time flies like an arrow..."))
  }
}
```

Förekomsten av en anonym funktion i detta exempel avslöjas av den högra pilen '=>' som separerar funktionens argumentlista från dess innehåll. I detta exempel är listan av argument tom, vilket går att urskilja utifrån det tomma paret av parenteser vänster om pilen. Funktionens innehåll är samma som för timeFlies ovanför.

# 5 Klasser

Som vi såg ovanför är Scala ett objektorienterat språk med ett koncept av klasser [2]. Klasser i Scala deklareras med en syntax liknande Javas syntax. En viktig skillnad är att klasser i Scala kan ta emot parametrar. Detta illustreras i följande defininion av komplexa tal.

```
class Complex(real: Double, imaginary: Double) {
  def re() = real
  def im() = imaginary
}
```

Denna klass tar emot två argument, vilket är de reella och imaginära delarna av ett komplext tal. Dessa argument måste skickas när en instans av klassen Complex skapas, som följande: **new** Complex(1.5, 2.3). Klassen innehåller två metoder, kallade re och im, vilka ger åtkomst till dessa båda delar.

Det bör noteras att returtypen av dessa två metoder inte ges explicit. Det kommer antydas (eng. *inferred*) automatiskt av kompilatorn, som tittar på den högra sidan av dessa metoder och härleder att båda returnerar ett värde av typen Double.

Kompilatorn kan inte alltid antyda typer så här, och det finns tyvärr ingen enkel regel för att veta exakt när det kommer bli så, och när det inte kommer bli så. I praktiken är detta inte vanligtvis något problem, då kompilatorn säger ifrån när den inte kan antyda en typ som inte uppgavs explicit. Som en enkel regel bör nybörjare inom Scala försöka utelämna typdeklarationer som verkar vara enkla att härleda från sammanhanget, och se om kompilatorn accepterar det. Efter ett tag bör programmeraren få en känsla för när typer ska uteslutas, och när de skall specificeras explicit.

## 5.1 Metoder utan argument

Ett mindre problem med metoderna re och im är att, för att kunna anropa dem, måste man använda ett par tomma parenteser efter deras namn, som följande exempel visar:

```
object ComplexNumbers {
  def main(args: Array[String]) {
    val c = new Complex(1.2, 3.4)
```

[2] För fullständighetens skull bör det noteras att somliga objektorienterade språk inte har något koncept av klasser, men att Scala inte är ett av dem.

```
        println("imaginary part: " + c.im())
    }
}
```

Det skulle vara trevligt om vi kunde komma åt de reella och imaginära delarna som om de vore fält, utan att behöva använda ett par tomma parenteser. Detta är genomförbart i Scala, helt enkelt genom att definiera de som metoder *utan argument*. Sådana metoder skiljer sig från metoder med inga argument på så sätt att de inte har parenteser efter deras namn, varken i definitionen eller när de används. Vår klass Complex kan skrivas om på följande sätt:

```
class Complex(real: Double, imaginary: Double) {
    def re = real
    def im = imaginary
}
```

### 5.2 Arv och överskuggning

Alla klasser i Scala ärver från en superklass. När ingen superklass specificeras, som i exemplet med Complex i föregående sektion, används scala.AnyRef implicit.

Det är möjligt att överskugga metoder ärvda från en superklass i Scala. Däremot är det obligatoriskt att explicit specificera att en metod överskuggar en annan genom att använda modifieraren **override**, för att undvika oavsiktlig överskuggning. Som ett exempel kan vår klass Complex utökas med en omdefinition av den ärvda metoden toString från Object.

```
class Complex(real: Double, imaginary: Double) {
    def re = real
    def im = imaginary
    override def toString() =
        "" + re + (if (im < 0) "" else "+") + im + "i"
}
```

## 6 Case klasser och mönsterigenkänning

En slags datastruktur som ofta förekommer i program är träd. Exempelvis interpreterare och kompilatorer representerar vanligtvis program internt i form av träd; XML-dokument är träd; och åtskilliga typer av behållare (eng. *container*) baseras på träd, såsom röd-svarta träd (eng. *red-black trees*).

Vi ska nu undersöka hur sådana träd representeras och manipuleras i Scala genom en simpel kalkylator. Syftet med programmet är att manipulera väldigt enkla aritmetiska uttryck bestående av summor, heltalskonstanter och variabler.
Två exempel på sådana uttryck är 1 +2 och (*x* +*x*) +(7 + *y*).
Vi måste först bestämma oss för hur sådana uttryck skall representeras. Det

naturligaste valet är ett träd, där noder är operationer (här, adderingen) och löv är värden (här konstanter eller variabler).

I Java skulle ett sådant träd representeras m h a en abstrakt superklass för träden, och en konkret subklass per nod eller löv. I ett funktionellt programmeringsspråk skulle en algebraisk datatyp användas för samma ändamål. Scala tillhandahåller ett koncept av s.k. *case klasser* (eng. *case classes*), vilket är något däremellan. Så här kan de användas för att definiera typen av träd för vårt exempel:

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

Det faktum att klasserna Sum, Var och Const deklareras att vara case klasser betyder att de skiljer sig åt från vanliga klasser i flera avseenden:

- Nyckelordet **new** behövs inte för att skapa instanser av dessa klasser (d.v.s. att det går att skriva Const(5) istället för **new** Const(5)),
- getter funktioner definieras automatiskt för konstruktorparametrar (d.v.s. att det är möjligt att hämta värden från konstruktorparametern v av samma instans c av klassen Const genom att helt enkelt skriva c.v),
- Standarddefinitioner för metoderna equals och hashCode är tillhandahållna, vilket fungerar på *strukturen* av instanserna och inte på deras identitet,
- En standarddefinition för metoden toString är tillhandahållen, och skriver ut ett värde i källkodformat (t.ex. trädet för uttrycket *x*+1 skrivs ut som Sum(Var(x),Const(1))),
- Instanserna av dessa klasser kan brytas ned genom *mönsterigenkänning* (eng. *pattern matching*), som vi kommer se nedanför.

Eftersom vi har definierat datatypen att representerar våra aritmetiska uttryck, kan vi börja definiera operationer för att manipulera dem. Vi börjar med en funktion för att evaluera ett uttryck i någon slags *miljö*. Syftet med miljön är att ge värden till variabler. Som exempel kan vi ta uttrycket *x* +1 som evalueras i en miljö som associerar värdet 5 med variabeln *x*, vilket skrivs $\{x \rightarrow 5\}$, vilket ger resultatet 6.
Vi måste således hitta ett sätt att representera miljöer. Vi skulle förstås kunna använda någon associativ datastruktur såsom en hashtabell, men vi kan även använda funktioner! En miljö är inget annat än en funktion som associerar ett värde till ett (variabel) namn. Miljön $\{x \rightarrow 5\}$ ovanför kan helt enkelt skrivas på följande sätt i Scala:

```
{ case "x" => 5 }
```

Denna notation definierar en funktion som, givet en sträng ”x” som argument, returnerar heltalet 5, och i andra fall kastar ett undantag.

Låt oss ge ett namn till de olika typerna av miljöer innan vi skriver evalueringsfunktionen. Vi skulle förstås alltid kunna använda typen String => Int för miljöer, men det förenklar programmet ifall vi introducerar ett namn för denna typ, och gör även framtida ändringar enklare. Detta uppnås i Scala med följande notation:

```
type Environment = String => Int
```

Från och med då kan typen Environment användas som ett alias för typerna av funktioner från String till Int.

Vi kan nu ge en definition av evalueringsfunktionen. Konceptuellt sätt är det väldigt enkelt: värdet av summan av två uttryck är helt enkelt summan av värdet av dessa uttryck; värdet av en variabel erhålls direkt från miljön; och värdet av en konstant är själva konstanten. Detta uttryckt i Scala inte svårare än så:

```
def eval(t: Tree, env: Environment): Int = t match {
      case Sum(l, r) => eval(l, env) + eval(r, env)
      case Var(n)    => env(n)
      case Const(v)  => v
    }
```

Denna evalueringsfunktion fungerar genom att utföra *mönsterigenkänning* på trädet t. Intuitivt sätt bör betydelsen av ovanstående definition vara tydligt:

1. den kollar först ifall trädet t är av typen Sum. Om så är fallet binder den det vänstra subträdet till en ny variabel kallad l och det högra subträdet till en variabel kallad r, och fortsätter sedan med att evaluera uttrycket efter pilen; detta uttryck kan och gör användning av variablerna bundna av mönstret som förekommer vänster om pilen, d.v.s. l och r,

2. om den första kontrollen misslyckas, d.v.s. om trädet inte är av typen Sum, går den vidare med att kolla om t är av typen Var. Om så är fallet binder den namnet i noden Var till en variabel n och fortsätter med det högra uttrycket,

3. om den andra kontrollen även den misslyckas, d.v.s. om t varken är av typen Sum eller Var, så kontrollerar den om det är en Const, och om så är fallet, så binder den värdet i noden Const till en variabel v och fortsätter med den högra sidan,

4. till sist, om alla tidigare kontroller misslyckas kommer ett undantag kastas för att signalera att mönsterigenkänningsuttrycket misslyckats; detta kan endast hända om flera subklasser av Tree har deklarerats.

Vi ser att den grundläggande idén bakom mönsterigenkänning är att försöka matcha ett värde mot en serie av mönster, och så fort ett mönster matchar, så extraheras och namnges diverse delar av värdet, för att till sist evaluera en del kod som vanligtvis använder sig av dessa namngivna delar.

En erfaren objektorienterad programmerare kanske undrar varför vi inte definierade eval att vara en metod av klassen Tree och dess subklasser. Vi skulle kunna ha gjort så, då Scala tillåter metoddefinitioner i case klasser precis som i vanliga klasser. Att avgöra huruvida mönsterigenkänning eller metoder skall användas är därför upp till var och en, men det har viktiga implikationer vad gäller utvidgning (eng. *extensibility*):

- när metoder används är det enkelt att lägga till en ny typ av nod då detta kan göras genom att definiera den att vara en subklass av Tree; å andra sidan är det tröttsamt att lägga till nya operationer för att manipulera trädet, då det kräver att alla subklasser av Tree måste modifieras,
- när mönsterigenkänning används är situationen det omvända: att lägga till en ny typ av nod kräver att alla funktioner som gör mönsterigenkänning på trädet måste modifieras för att kunna ta hänsyn till den nya noden; å andra sidan är det enkelt att lägga till en ny operation, genom att definiera det som en fristående funktion.

Vi fortsätter undersöka mönsterigenkänning genom att definiera en annan operation på aritmetiska uttryck: symbolisk derivering. Läsaren kanske kommer ihåg följande regler vad gäller denna operation:

1. derivatan av en summa är summan av alla derivata,
2. derivatan av någon variabel *v* är ett om *v* är variabeln relativ var derivationen tar plats, och noll i andra fall,
3. derivatan av en konstant är noll.

Dessa regler kan nästan bokstavligen översättas till Scalakod, vilket ger upphov till följande definition:

```
def derive(t: Tree, v: String): Tree = t match {
  case Sum(l, r) => Sum(derive(l, v), derive(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

Denna funktion introducerar två nya koncept besläktade med mönsterigenkänning. För det första, uttrycket **case** för variabler har en s.k. vakt (eng. *guard*), ett uttryck som följer nyckelordet **if**. Denna vakt hindrar mönsterigenkänningen från att lyckas om inte dess uttryck är sant. Här används den för att se till att vi returnerar konstanten 1 enbart om namnet av den variabel som deriveras är samma som den

deriverade variabeln v. Den andra nya funktionen av mönsterigenkänning här är en s.k. *wild-card*, som skrivs _, vilket är ett mönster som matchar vilket värde som helst, utan att ge den ett namn.

Vi har ännu inte undersökt alla starka sidor hos mönsterigenkänning ännu, men vi ska avsluta här för att hålla detta dokument kort. Vi vill fortfarande se hur de två ovanstående funktionerna uppträder i ett faktiskt exempel. För detta ändamål skall vi nu skriva en simpel main funktion som utför några operationer på uttrycket $(x + x) + (7 + y)$: den beräknar först dess värde i miljön $\{x \rightarrow 5, y \rightarrow 7\}$, och beräknar sedan derivatorna relativt $x$ och därefter $y$.

```
def main(args: Array[String]) {
  val exp: Tree = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
  val env: Environment = { case "x" => 5 case "y" => 7 }
  println("Expression: " + exp)
  println("Evaluation with x=5, y=7: " + eval(exp, env))
  println("Derivative relative to x:\n " + derive(exp, "x"))
  println("Derivative relative to y:\n " + derive(exp, "y"))
}
```

När vi exekverar detta program får vi följande förväntade utskrift:

```
Expression: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative to x:
 Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivative relative to y:
 Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))
```

Genom att undersöka utskriften ser vi att resultatet av derivatorna bör förenklas innan det presenteras för användaren. Att definiera en grundläggande funktion för att förenkla derivata m h a mönsterigenkänning är ett intressant (men förvånandevis knepigt) problem, som vi lämnar som övning till läsaren.

# 7 Traits

Bortsett från att ärva kod från en superklass, så kan en klass i Scala även importera kod från eller flera s.k. *traits*.

Kanske det enklaste sättet för en Javaprogrammerare att förstå vad traits är, är att betrakta dem som gränssnitt som även kan innehålla kod. När en klass ärver från en trait i Scala så implementerar den en traits gränssnitt, och ärver all kod från denna trait.

Låt oss se på ett klassiskt exempel för att undersöka en traits användbarhet: nämligen ordnade objekt. Det är ofta användbart att kunna jämföra objekt av en

given klass sinsemellan, genom att t.ex. sortera dem. Objekt i Java som är jämförbara implementerar gränssnittet Comparable. I Scala kan vi åstadkomma något bättre än vad vi skulle i Java genom att definiera en motsvarande Comparable att vara en trait, som vi skall kalla för Ord.

När objekt jämförs finns det sex stycken olika predikat som kan vara användbara: mindre, mindre eller lika med, lika med, inte lika med, större än eller lika med, och större än. Emellertid är det besvärligt att definiera alla, speciellt med tanke på att fyra av dessa sex kan uttryckas med de två återstående. Givet lika eller mindre predikat (till exempel), så går det att uttrycka de andra. I Scala kan alla dessa observationer fångas upp med följande deklaration av en trait:

```
trait Ord {
  def < (that: Any): Boolean
  def <=(that: Any): Boolean = (this < that) || (this == that)
  def > (that: Any): Boolean = !(this <= that)
  def >=(that: Any): Boolean = !(this < that)
}
```

Denna definition både skapar en ny typ kallad Ord, som har samma roll som Javas gränssnitt Comparable har, och har även standardimplementationer av tre predikat i termer av en fjärde abstrakt. Predikaten för likhet och olikhet visar sig inte här då de som standard är närvarande i alla objekt.

Typen Any som används ovanför är den typ som är en supertyp för alla andra typer i Scala. Man kan betrakta det som en mer generell version av Javatypen Object, då även den är en supertyp av mer grundläggande typer, såsom Int, Float, etc.

För att göra objekt av en klass jämförbara så är det tillräckligt att definiera de predikat som testar likhet och mixa in klassen Ord ovanför. Som ett exempel, låt oss definiera en klass Date som representerar datum i en gregoriansk kalender. Sådana datum består av en dag, en månad, och ett år, som vi skall representera som heltal. Vi börjar därför definitionen av klassen Date som följande:

```
class Date(y: Int, m: Int, d: Int) extends Ord {
  def year = y
  def month = m
  def day = d

  override def toString(): String = year + "-" + month + "-" + day
```

Den viktiga delen här är deklarationen **extends** Ord som följer klassnamnet och parametrarna. Den säger att klassen Date ärver från vår trait Ord.

Därefter omdefinierar vi metoden equals, ärvd från Object, så att den jämför datum korrekt genom att jämföra deras individuella fält. Standard implementationen av equals är inte användbar, då den, precis som i Java, jämför objekt fysiskt. Vi kommer fram till följande definition:

```
override def equals(that: Any): Boolean =
  that.isInstanceOf[Date] && {
    val o = that.asInstanceOf[Date]
    o.day == day && o.month == month && o.year == year
  }
```

Denna metod använder de fördefinierade metoderna isInstanceOf och asInstanceOf. Den första, isInstanceOf, korresponderar mot Javaoperatorn instanceof, och returnerar true om och endast om objektet som den appliceras på är en instans av den givna typen. Den andra, asInstanceOf, korresponderar mot Javas typomvandlingsoperator: om objektet är en instans av den givna typen, så betraktas det som en sådan, annars kastas en ClassCastException.

Till sist, den sista metoden som behöver definieras är predikatet som testar om ett värde är mindre. Detta gör den genom att använda sig av en annan fördefinierad metod, error, som kastar ett undantag med ett givet felmeddelande:

```
def <(that: Any): Boolean = {
  if (!that.isInstanceOf[Date])
    error("cannot compare " + that + " and a Date")

  val o = that.asInstanceOf[Date] (year <
  o.year) ||
  (year == o.year && (month < o.month ||
                      (month == o.month && day < o.day)))
  }
```

Med detta avslutar vi definitionen av klassen Date. Instanser av denna klass kan betraktas antingen som datum eller som jämförbara objekt. Utöver detta definierar de allesammans sex jämförelsepredikat som nämns ovan: equals och < då de förekommer direkt i definitionen av klassen Date, och alla andra för att de ärvs från vår trait Ord.

Traits är förstås användbara i flera andra situationer än vad vi såg här. Vi nöjer oss med detta, då en längre diskussion av deras applikationer är alltför omfattande för det här dokumentet.

# 8 Generisk programmering

Det sista kännetecknet av Scala som vi kommer utforska i denna handledning är generisk programmering. Javaprogrammerare bör vara väl medvetna om dem

problem som uppstår med bristen på generiskhet i deras språk, en brist som åtgärdas i Java 1.5.

Generiskhet är förmågan att skriva parametriserad kod av olika typer. Som exempel kan vi ta en programmerare som skriver ett bibliotek för länkade listor. Problemet som uppstår är att avgöra vilken typ som ska tilldelas elementen i listan. Då listan är menad att användas i många olika sammanhang är det inte möjligt att avgöra att typen av element skall vara, låt oss säga, Int. Detta skulle vara helt och hållet godtyckligt och alltför restriktivt.

Javaprogrammerare tar till hjälp Object, vilket är supertypen av alla objekt. Denna lösning är däremot långt ifrån idealisk, då den inte fungerar för grundtyper (int, long, float, etc.) och det implicerar att en hel del dynamiska typomvandlingar måste sättas in av programmeraren.

Scala gör det möjligt att definiera generiska klasser (och metoder) för att lösa detta problem. Låt oss undersöka detta med ett exempel: en referens, som kan vara antingen tom eller peka på ett objekt av någon typ.

```
class Reference[T] {
  private var contents: T = _

  def set(value: T) { contents = value }
  def get: T = contents
}
```

Klassen Reference är parametriserad av en typ, kallad T, som är typen av dess element. Denna typ används i kroppen av klassen som typen av variabeln contents, argumenten i metoden set, och returtypen av metoden get.

Kodexemplet ovanför introducerar variabler i Scala, vilket inte bör behöva några ytterligare förklaringar. Det är dock intressant att se att startvärdet till variabeln är _, vilket representerar ett standardvärde. Detta standardvärde är 0 för numeriska typer, **false** för booleska typer, () för typen Unit, och **null** för alla objekttyper.

För att kunna använda klassen Reference måste man specificera vilken typ som skall användas för typparametern T, vilket är typen av elementet som finns i cellen. Som exempel kan vi skapa och använda en cell innehållande ett heltal, som kan skrivas på följande sätt:

```
object IntegerReference {
  def main(args: Array[String]) {
    val cell = new Reference[Int]
    cell.set(13)
```

```
        println("Reference contains the half of " + (cell.get * 2))
    }
}
```

Som går att urskilja från exemplet ovanför är det inte nödvändigt att omvandla värdet som returneras av metoden get innan den används som ett heltal. Det är även inte möjligt att lagra något annat än ett heltal i just denna cell, då den deklarerades att innehålla ett heltal.

# 9 Slutsats

Detta dokument gav en snabb överblick av språket Scala och presenterade några grundläggande exempel. Den nyfikna läsaren kan fortsätta med att läsa det uppföljande dokumentet *Scala By Example*, vilket innehåller mycket mer avancerade exempel, och konsultera *Scala Language Specification* när så behövs.